15.3 — Move constructors and move assignment

BY ALEX ON FEBRUARY 26TH, 2017 | LAST MODIFIED BY ALEX ON APRIL 4TH, 2017

In lesson **15.1 -- Intro to smart pointers and move semantics**, we took a look at std::auto_ptr, discussed the desire for move semantics, and took a look at some of the downsides that occur when functions designed for copy semantics (copy constructors and copy assignment operators) are redefined to implement move semantics.

In this lesson, we'll take a deeper look at how C++11 resolves these problems via move constructors and move assignment.

Copy constructors and copy assignment

First, let's take a moment to recap copy semantics.

Copy constructors are used to initialize a class by making a copy of an object of the same class. Copy assignment is used to copy one class to another existing class. By default, C++ will provide a copy constructor and copy assignment operator if one is not explicitly provided. These compiler-provided functions do shallow copies, which may cause problems for classes that allocate dynamic memory. So classes that deal with dynamic memory should override these functions to do deep copies.

Returning back to our Auto_ptr smart pointer class example from the first lesson in this chapter, let's look at a version that implements a copy constructor and copy assignment operator that do deep copies, and a sample program that exercises them:

```
1  #include <iostream>
2
3  template<class T>
4  class Auto_ptr3
5  {
6      T* m_ptr;
7  public:
8      Auto_ptr3(T* ptr = nullptr)
9          :m_ptr(ptr)
10     {
11     }
12
13     ~Auto_ptr3()
14     {
15         delete m_ptr;
16     }
17
18     // Copy constructor
19     // Do deep copy of a.m_ptr to m_ptr
       Auto_ptr3(const Auto_ptr3& a)
20     {
21         m_ptr = new T;
22         *m_ptr = *a.m_ptr;
23     }
24
25     // Copy assignment
26     // Do deep copy of a.m_ptr to m_ptr
27     Auto_ptr3& operator=(const Auto_ptr3& a)
       {
28         // Self-assignment detection
           if (&a == this)
29             return *this;
30
```

```
        // Release any resource we're holding
        delete m_ptr;

        // Copy the resource
        m_ptr = new T;
        *m_ptr = *a.m_ptr;

        return *this;
    }

    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
    bool isNull() const { return m_ptr == nullptr; }
};

class Resource
{
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};

Auto_ptr3<Resource> generateResource()
{
    Auto_ptr3<Resource> res(new Resource);
    return res; // this return value will invoke the copy constructor
}

int main()
{
    Auto_ptr3<Resource> mainres;
    mainres = generateResource(); // this assignment will invoke the copy assignment

    return 0;
}
```

In this program, we're using a function named generateResource() to create a smart pointer encapsulated resource, which is then passed back to function main(). Function main() then assigns that to an existing Auto_ptr3 object.

When this program is run, it prints:

```
Resource acquired
Resource acquired
Resource destroyed
Resource acquired
```

```
Resource destroyed
Resource destroyed
```

That's a lot of resource creation and destruction going on for such a simple program! What's going on here?

Let's take a closer look. There are 6 key steps that happen in this program (one for each printed message):

1) Inside generateResource(), local variable res is created and initialized with a dynamically allocated Resource, which causes the first "Resource acquired".
2) Res is returned back to main() by value. We return by value here because res is a local variable -- it can't be returned by address or reference because res will be destroyed when generateResource() ends. So res is copy constructed into a temporary object. Since our copy constructor does a deep copy, a new Resource is allocated here, which causes the second "Resource acquired".
3) Res goes out of scope, destroying the originally created Resource, which causes the first "Resource destroyed".
4) The temporary object is assigned to mainres by copy assignment. Since our copy assignment also does a deep copy, a new Resource is allocated, causing yet another "Resource acquired".
5) The assignment expression ends, and the temporary object goes out of expression scope and is destroyed, causing a "Resource destroyed".
6) At the end of main(), mainres goes out of scope, and our final "Resource destroyed" is displayed.

So, in short, because we call the copy constructor once to copy construct res to a temporary, and copy assignment once to copy the temporary into mainres, we end up allocating and destroying 3 separate objects in total.

Inefficient, but at least it doesn't crash!

However, with move semantics, we can do better.

Move constructors and move assignment

C++11 defines two new functions in service of move semantics: a move constructor, and a move assignment operator. Whereas the goal of the copy constructor and copy assignment is to make a copy of one object to another, the goal of the move constructor and move assignment is to move the resources from one object to another.

Defining a move constructor and move assignment work analogously to their copy counterparts. However, whereas the copy flavors of these functions take a const l-value reference parameter, the move flavors of these functions use non-const r-value reference parameters.

Here's the same Auto_ptr3 class as above, with a move constructor and move assignment operator added. We've left in the deep-copying copy constructor and copy assignment operator for comparison purposes.

```
1  #include <iostream>
2
3  template<class T>
4  class Auto_ptr4
5  {
6      T* m_ptr;
7  public:
8      Auto_ptr4(T* ptr = nullptr)
9          :m_ptr(ptr)
10     {
11     }
12
13     ~Auto_ptr4()
14     {
15         delete m_ptr;
16     }
17
18     // Copy constructor
19     // Do deep copy of a.m_ptr to m_ptr
```

```
    Auto_ptr4(const Auto_ptr4& a)
    {
        m_ptr = new T;
        *m_ptr = *a.m_ptr;
    }

    // Move constructor
    // Transfer ownership of a.m_mptr to m_ptr
    Auto_ptr4(Auto_ptr4&& a)
        : m_ptr(a.m_ptr)
    {
        a.m_ptr = nullptr;
    }

    // Copy assignment
    // Do deep copy of a.m_ptr to m_ptr
    Auto_ptr4& operator=(const Auto_ptr4& a)
    {
        // Self-assignment detection
        if (&a == this)
            return *this;

        // Release any resource we're holding
        delete m_ptr;

        // Copy the resource
        m_ptr = new T;
        *m_ptr = *a.m_ptr;

        return *this;
    }

    // Move assignment
    // Transfer ownership of a.m_ptr to m_ptr
    Auto_ptr4& operator=(Auto_ptr4&& a)
    {
        // Self-assignment detection
        if (&a == this)
            return *this;

        // Release any resource we're holding
        delete m_ptr;

        // Transfer ownership of a.m_ptr to m_ptr
        m_ptr = a.m_ptr;
        a.m_ptr = nullptr;

        return *this;
    }

    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
    bool isNull() const { return m_ptr == nullptr; }
};

class Resource
{
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};
```

```
71  Auto_ptr4<Resource> generateResource()
    {
72      Auto_ptr4<Resource> res(new Resource);
        return res; // this return value will invoke the move constructor
73  }
74
75  int main()
76  {
77      Auto_ptr4<Resource> mainres;
78      mainres = generateResource(); // this assignment will invoke the move assignment

79      return 0;
    }
80
81
82


83
84


85



86
87
88
89
90
91



92
93
94
```

The move constructor and move assignment operator are simple. Instead of deep copying the source object (a) into the implicit object, we simply move (steal) the source object's resources. This involves shallow copying the source pointer into the implicit object, then setting the source pointer to null.

When run, this program prints:

```
Resource acquired
Resource destroyed
```

That's much better!

The flow of the program is exactly the same as before. However, instead of calling the copy constructor and copy assignment operators, this program calls the move constructor and move assignment operators. Looking a little more deeply:

1) Inside generateResource(), local variable res is created and initialized with a dynamically allocated Resource, which causes the first "Resource acquired".
2) Res is returned back to main() by value. Res is move constructed into a temporary object, transferring the dynamically created object stored in res to the temporary object.
3) Res goes out of scope. Because res no longer manages a pointer (it was moved to the temporary), this is a non-event.
4) The temporary object is move assigned to mainres. This transfers the dynamically created object stored in the temporary to mainres.
5) The assignment expression ends, and the temporary object goes out of expression scope and is destroyed. However,

because the temporary no longer manages a pointer (it was moved to mainres), this is also a non-event.
6) At the end of main(), mainres goes out of scope, and our final “Resource destroyed” is displayed.

So instead of copying our Resource twice (once for the copy constructor and once for the copy assignment), we transfer it twice. This is more efficient, as Resource is only constructed and destroyed once instead of three times.

In most cases, a move constructor and move assignment operator will not be provided by default, unless the class does not have any define copy constructors, copy assignment, move assignment, or destructors. However, the default move constructor and move assignment do the same thing as the default copy constructor and copy assignment (make copies, not do moves).

Rule: If you want a move constructor and move assignment that do moves, you’ll need to write them yourself.

When are the move constructor and move assignment called?

The move constructor and move assignment are called when those functions have been defined, and the argument for construction or assignment is an r-value. Most typically, this r-value will be a literal or temporary value.

The key insight behind move semantics

You now have enough context to understand the key insight behind move semantics.

If we construct an object or do an assignment where the argument is an l-value, the only thing we can reasonably do is copy the l-value. We can’t assume it’s safe to alter the l-value, because it may be used again later in the program. If we have an expression “a = b”, we wouldn’t reasonably expect b to be changed in any way.

However, if we construct an object or do an assignment where the argument is an r-value, then we know that r-value is just a temporary object of some kind. Instead of copying it (which can be expensive), we can simply transfer its resources (which is cheap) to the object we’re constructing or assigning. This is safe to do because the temporary will be destroyed at the end of the expression anyway, so we know it will never be used again!

C++11, through r-value references, gives us the ability to provide different behaviors when the argument is an r-value vs an l-value, enabling us to make smarter and more efficient decisions about how our objects should behave.

Disabling copying

In the Auto_ptr4 class above, we left in the copy constructor and assignment operator for comparison purposes. But in move-enabled classes, it is sometimes desirable to delete the copy constructor and copy assignment functions to ensure copies aren’t made. In the case of our Auto_ptr class, we don’t want to copy our templated object T -- both because it’s expensive, and whatever class T is may not even support copying!

Here’s a version of Auto_ptr that supports move semantics but not copy semantics:

```
1  #include <iostream>
2
3  template<class T>
4  class Auto_ptr5
5  {
6      T* m_ptr;
7  public:
8      Auto_ptr5(T* ptr = nullptr)
9          :m_ptr(ptr)
10     {
11     }
12
13     ~Auto_ptr5()
14     {
15         delete m_ptr;
16     }
17
18     // Copy constructor -- no copying allowed!
```

```
    Auto_ptr5(const Auto_ptr5& a) = delete;

    // Move constructor
    // Transfer ownership of a.m_mptr to m_ptr
    Auto_ptr5(Auto_ptr5&& a)
        : m_ptr(a.m_ptr)
    {
        a.m_ptr = nullptr;
    }

    // Copy assignment -- no copying allowed!
    Auto_ptr5& operator=(const Auto_ptr5& a) = delete;

    // Move assignment
    // Transfer ownership of a.m_ptr to m_ptr
    Auto_ptr5& operator=(Auto_ptr5&& a)
    {
        // Self-assignment detection
        if (&a == this)
            return *this;

        // Release any resource we're holding
        delete m_ptr;

        // Transfer ownership of a.m_ptr to m_ptr
        m_ptr = a.m_ptr;
        a.m_ptr = nullptr;

        return *this;
    }

    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
    bool isNull() const { return m_ptr == nullptr; }
};
```

If you were to try to pass an Auto_ptr5 l-value to a function by value, the compiler would complain that the copy constructor required to initialize the copy constructor argument has been deleted. This is good, because we should probably be passing Auto_ptr5 by const l-value reference anyway!

Auto_ptr5 is (finally) a good smart pointer class. And, in fact the standard library contains a class very much like this one (that you should use instead), named std::unique_ptr. We'll talk more about std::unique_ptr later in this chapter.

Another example

Let's take a look at another class that uses dynamic memory: a simple dynamic templated array. This class contains a deep-copying copy constructor and copy assignment operator.

```
#include <iostream>
```

```
2
3   template <class T>
4   class DynamicArray
5   {
6   private:
7       T* m_array;
8       int m_length;
9
10  public:
11      DynamicArray(int length)
12          : m_length(length), m_array(new T[length])
        {
13      }
14
15      ~DynamicArray()
16      {
17          delete[] m_array;
18      }
19
20      // Copy constructor
21      DynamicArray(const DynamicArray &arr)
22          : m_length(arr.m_length)
        {
23          m_array = new T[m_length];
24          for (int i = 0; i < m_length; ++i)
25              m_array[i] = arr.m_array[i];
26      }

27      // Copy assignment
        DynamicArray& operator=(const DynamicArray &arr)
28      {
29          if (&arr == this)
30              return *this;
31
            delete[] m_array;
32
33          m_length = arr.m_length;
34          m_array = new T[m_length];
35
36          for (int i = 0; i < m_length; ++i)
37              m_array[i] = arr.m_array[i];
38
39          return *this;
40      }
41
        int getLength() const { return m_length; }
42      T& operator[](int index) { return m_array[index]; }
        const T& operator[](int index) const { return m_array[index]; }
43
44  };
45
46
47


48


49


50
51
```

Now let's use this class in a program. To show you how this class performs when we allocate a million integers on the heap, we're going to leverage a new class, named Timer. We'll use the Timer class to time how fast our code runs, and show you the performance difference between copying and moving.

```
#include <iostream>
#include <chrono> // for std::chrono functions

// Uses the above DynamicArray class

class Timer
{
private:
    // Type aliases to make accessing nested type easier
    using clock_t = std::chrono::high_resolution_clock;
    using second_t = std::chrono::duration<double, std::ratio<1> >;

    std::chrono::time_point<clock_t> m_beg;

public:
    Timer() : m_beg(clock_t::now())
    {
    }

    void reset()
    {
        m_beg = clock_t::now();
    }

    double elapsed() const
    {
        return std::chrono::duration_cast<second_t>(clock_t::now() - m_beg).count();
    }
};

// Return a copy of arr with all of the values doubled
DynamicArray<int> cloneArrayAndDouble(const DynamicArray<int> &arr)
{
    DynamicArray<int> dbl(arr.getLength());
    for (int i = 0; i < arr.getLength(); ++i)
        dbl[i] = arr[i] * 2;

    return dbl;
}

int main()
{
    Timer t;

    DynamicArray<int> arr(1000000);

    for (int i = 0; i < arr.getLength(); i++)
        arr[i] = i;

    arr = cloneArrayAndDouble(arr);

    std::cout << t.elapsed();
}
```

```
45
46
47
48
49
50
51
52
53
```

On one of the author's machines, in release mode, this program executed in 0.00825559 seconds.

Now let's run the same program again, replacing the copy constructor and copy assignment with a move constructor and move assignment.

```
1  template <class T>
2  class DynamicArray
3  {
4  private:
5      T* m_array;
6      int m_length;
7
8  public:
9      DynamicArray(int length)
10         : m_length(length), m_array(new T[length])
       {
11     }
12
13     ~DynamicArray()
14     {
15         delete[] m_array;
16     }
17
18     // Copy constructor
19     DynamicArray(const DynamicArray &arr) = delete;
20
       // Copy assignment
21     DynamicArray& operator=(const DynamicArray &arr) = delete;
22
23     // Move constructor
       DynamicArray(DynamicArray &&arr)
24         : m_length(arr.m_length), m_array(arr.m_array)
25     {
26         arr.m_length = 0;
           arr.m_array = nullptr;
27     }
28
29     // Move assignment
30     DynamicArray& operator=(DynamicArray &&arr)
31     {
32         if (&arr == this)
33             return *this;
34
           delete[] m_array;
35
           m_length = arr.m_length;
36         m_array = arr.m_array;
37         arr.m_length = 0;
38         arr.m_array = nullptr;
```

```
39
40          return *this;
41      }
42
43      int getLength() const { return m_length; }
44      T& operator[](int index) { return m_array[index]; }
45      const T& operator[](int index) const { return m_array[index]; }
46
47  };
48
49  #include <iostream>
    #include <chrono> // for std::chrono functions
50
    class Timer
51  {
    private:
        // Type aliases to make accessing nested type easier
52      using clock_t = std::chrono::high_resolution_clock;
53      using second_t = std::chrono::duration<double, std::ratio<1> >;
54
55      std::chrono::time_point<clock_t> m_beg;
56
    public:
57      Timer() : m_beg(clock_t::now())
58      {
59      }
60
61      void reset()
        {
62          m_beg = clock_t::now();
        }

63      double elapsed() const
        {
            return std::chrono::duration_cast<second_t>(clock_t::now() - m_beg).count();
64      }
65  };

66  // Return a copy of arr with all of the values doubled
67  DynamicArray<int> cloneArrayAndDouble(const DynamicArray<int> &arr)
68  {
        DynamicArray<int> dbl(arr.getLength());
69      for (int i = 0; i < arr.getLength(); ++i)
70          dbl[i] = arr[i] * 2;
71
72      return dbl;
73  }
74
75  int main()
76  {
77      Timer t;
78
79      DynamicArray<int> arr(1000000);

        for (int i = 0; i < arr.getLength(); i++)
80          arr[i] = i;
81
82      arr = cloneArrayAndDouble(arr);
83
        std::cout << t.elapsed();
84  }
```

```
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
```

On the same machine, this program executed in 0.0056 seconds.

Comparing the runtime of the two programs, 0.0056 / 0.00825559 = 67.8%. The move version was almost 33% faster!

15.4 -- std::move

Index

15.2 -- R-value references

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

9 comments to 15.3 — Move constructors and move assignment

Jason
April 24, 2017 at 12:33 pm · Reply

Hi alex, 2 questions:

1) at the end of 'Disabling copying' section you write:

"This is good, because we should probably be passing Auto_ptr5 by const l-value reference anyway!"
What do you mean by this?

2) Consider the code snippet:

```
Auto_ptr5& operator=(Auto_ptr5&& a)
{
    // Self-assignment detection
    if (&a == this)
        return *this;

    // Release any resource we're holding
    delete m_ptr;
```

My question is why do the self-assignment detection when &a /= this??

Tnanx Jason.

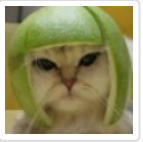
Alex
April 24, 2017 at 7:49 pm · Reply

1) We don't want to pass Auto_ptr5 by value, both because it invokes copy semantics, and because it's inefficient. If we want to pass an Auto_ptr5 to a function without having the function take ownership, we should be passing it in by reference (l-value reference, not r-value reference).

2) So that if the caller does something like w = std::move(w); m_ptr isn't deleted and your class is left in a consistent state.

rymcymcym
April 12, 2017 at 6:24 am · Reply

Could you expllain me why in Auto_ptr4, move assignment has been called? The res object is l_value, so why this temporary object was created with r_value?



Alex
April 14, 2017 at 10:39 am · Reply

res is an l-value, but the return value of the function is being returned by value. This return value is an r-value, since it is just a temporary.

SU
April 6, 2017 at 8:24 pm · Reply

I am trying to prove the default move assignment will not be provided if I have defined copy constructor or copy assignment or destructor, I modified an example online, which has defined copy constructor and destructor. But in main function when I try to use move assignment to do e1 = Example5("d"), the compiler does not complain anything, and the program ran well.

So this is obviously contradict to the statement "a move constructor and move assignment operator will not be provided by default, unless the class does not have any define copy constructors, copy assignment, move assignment, or destructors."

```
1 #include<iostream>
```

```
using namespace std;

class Example5 {
    string* ptr;
public:
    Example5 (const string& str) : ptr(new string(str)) {}
    ~Example5 () {delete ptr;}
    // copy constructor:
    Example5 (const Example5& x) : ptr(new string(x.content())) {}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {

    Example5 e1("w");
    e1 = Example5("d");

    return 0;
}
```

Alex
April 7, 2017 at 11:55 am · Reply

No, it's not contradicting. Your program, not finding a move assignment operator, is defaulting to copy assignment (which is provided by default).

Xiao Hu
April 3, 2017 at 9:02 pm · Reply

I think the implement of Move assignment in DynamicArray has some problems:
you don't free the memory of original m_array, you might use delete[] m_array.
Also, I think that a nice code can be:

```
DynamicArray& operator=(DynamicArray &&arr)
    {
        if (&arr == this)
            return *this;

        std::swap(m_length, arr.m_length);
        std::swap(m_array, arr.m_array);

        return *this;
    }
```

Alex
April 4, 2017 at 2:23 pm · Reply

Whoops, fixed the issue with the missing delete[]. Thanks for pointing that out.

Using std::swap here only works if the implicit array was originally empty. Otherwise, parameter arr ends up with the original array's length and a dangling pointer. That's okay if arr was really an r-value that was going out of scope, but if arr was a l-value that had been moved into an r-value, then we'd expect it to end up empty, not invalid.

Xiao Hu
April 4, 2017 at 8:17 pm · Reply

I still think that using std::swap has advantage:
uisng std::swap, the T* object of *this and arr is swapped. Regardless of the arr is a real l-value or r-value, the arr is destoryed once it goes out of scope. So, the orignal T* object of *this is destoryed. Using std::swap, we don't need to care how to destory the T* object.
Another problem is that the running time of move version is longer than the one of copy version. In fact, the result in my machine is instead opposite.